NTNU – Trondheim
Norwegian University of
Science and Technology

Tutorial Lecture for Exercise 1
TDT4258 Energy Efficient Computer Systems

Stefano Nichele
Department of Computer and Information Science
2013, January 25th

Practical Information

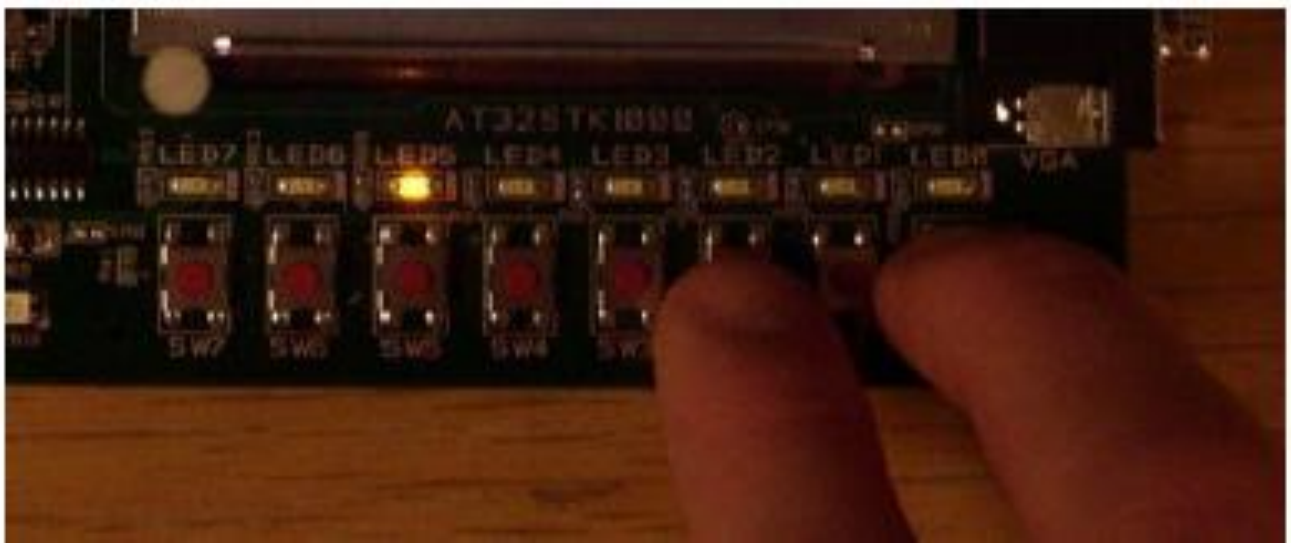
- Read the booklet (you can find it on the course wiki)
- Exercises:
  - Exercise 1: AVR32 Assembly. Buttons and LEDs.
  - Exercise 2: C on the AVR32. Sound.
  - Exercise 3: Linux on AVR32. Game.
- STK1000: be kind to our cards
  - Static electricity
  - Mechanical stress
- The Lab is available outside the scheduled lab hours

Practical Information

- Submission on It’s Learning
- Email to the assistant in case of emergency
- The report should be
  - in English
  - in PDF format
- Remember that the exercises are graded and copying is like cheating

Exercise 1

- Create a program that turns on the central LED and moves the light to the right or to the left depending on the pushed button (button 0: right, button 2: left)



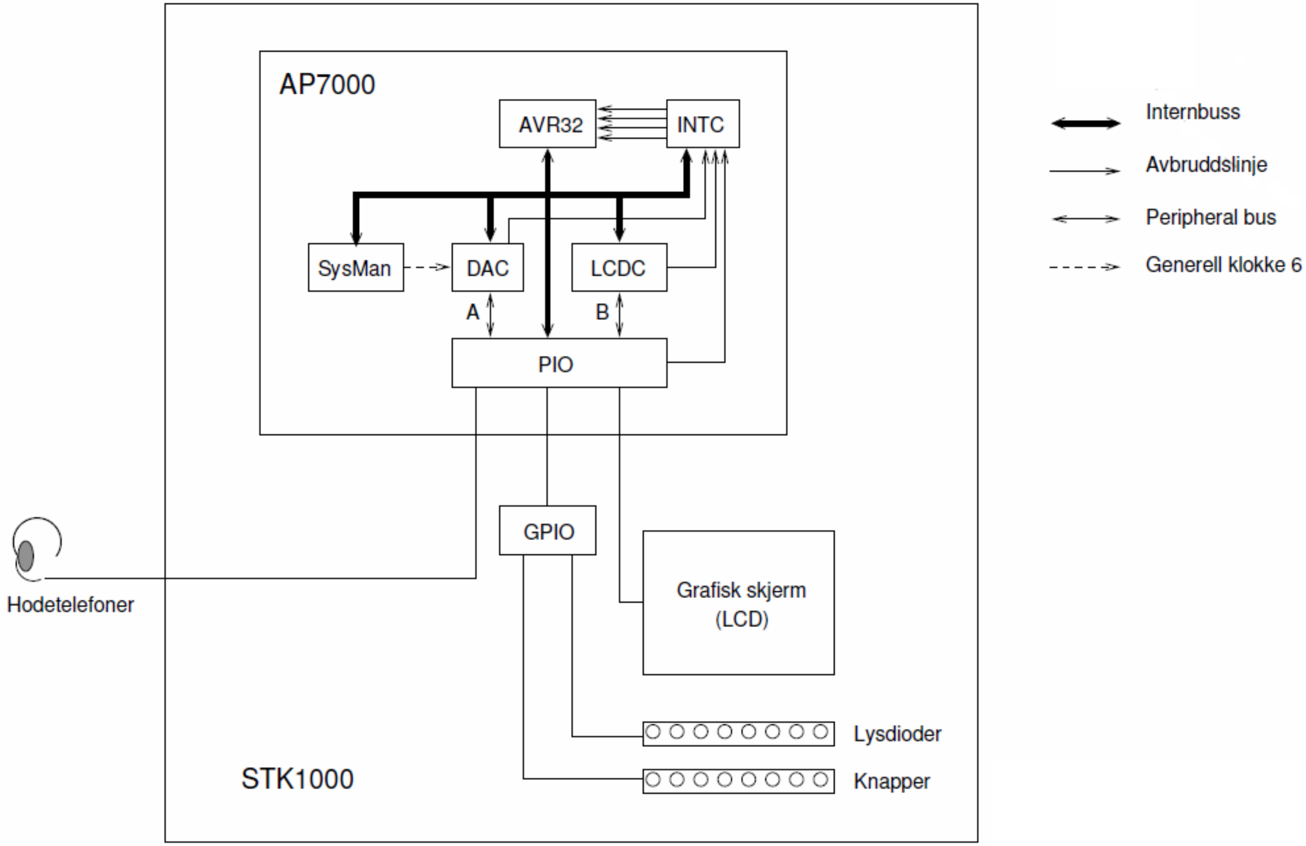
Requirements

- To be written in assembly language
- The buttons should be read in an interrupt routine
- The LEDs are updated in the main loop of the program
- You should use the GNU tools
  - GNU Assembler (GAS) and GNU Linker (LD)
  - Use the makefile
  - Debug with GDB

AVR32 and STK1000

- AVR32: 32-bit processor architecture, RISC load/store
- AT32AP7000: microcontroller with AVR32 processor
- STK1000: Development board with AT32AP7000

System overview

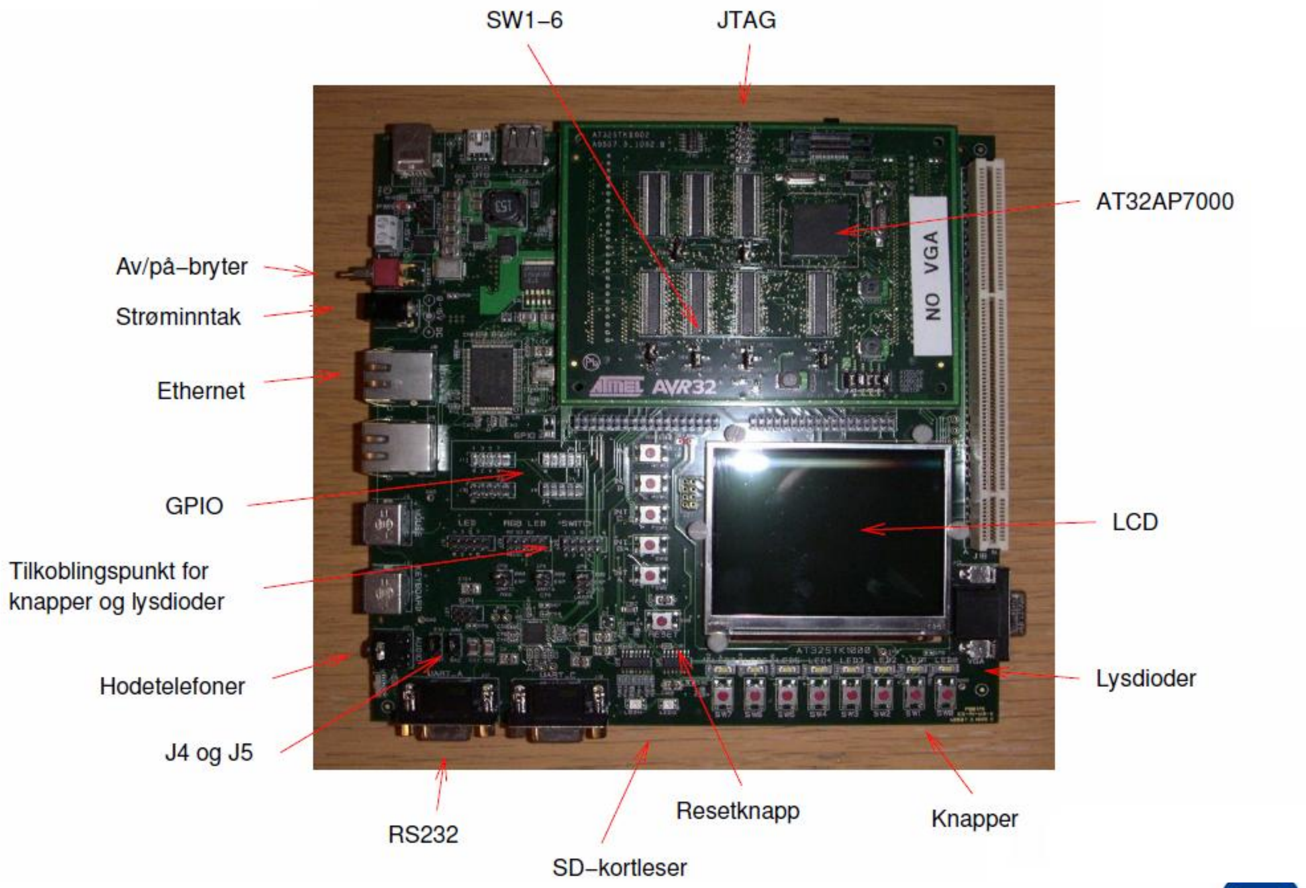


NTNU – Trondheim
Norwegian University of
Science and Technology

AVR32

- 32 bit
- Registry: 16 registers
  - 13 general: r0 – r12
  - Link Register: lr
  - Stack Pointer: sp
  - Program Counter: pc
- Many system registers, including:
  - Status register
  - EVBA

STK1000


SW1–6
JTAG
AT32AP7000
Av/på–bryter
Strøminntak
Ethernet
GPIO
LCD
Tilkoblingspunkt for
knapper og lysdioder
Hodetelefoner
Lysdioder
J4 og J5
RS232
SD–kortleser
Resetknapp
Knapper

JTAGICE mkII

Upload to STK1000

- To upload a program to STK1000 via JTAGICE use the following command:
  - Avr32program halt
  - Avr32program program –e –f0,8Mb <programfile>

AVR32 assembler

- Instructions <instruction name> <arguments>

```
mov r1,r0
```

- Comments /* This is a comment */

- Several types of symbols:
  - <symbolname> = <value>
  - <symbolname> : <instruction>

```
NINJA = 0xBEEF
loop:    sub r0,1
         brne loop
```

AVR32 assembler

- Arithmetic: ADD, SUB etc.

  ADD r1,r4 set r1 = r1 + r4
- Memory access: LD.*size*. ST.*size*

  LD.W r0,r1 download a word (32-bit) from the memory address located in r1 into r0
- Jump: BR*condition*

  BREQ jumptarget jump to the target if the Z flag is set

NTNU – Trondheim
Norwegian University of Science and Technology

AVR32 assembler

- Pseudo instructions:

  .include ”filename”

  .globl symbol

  .text

  .data

Segments

- Machine code is divided into segments
  - text program code, cannot be modified
  - data variables
- Pseudo instructions .text or .data indicate the segment for the subseqent code

Setup of assembler file

```
/* explicitly set the symbols */

.text

.globl  _start
_start:

/* program code */

.data

/* data areas*/
```

Parallel I/O: PIO

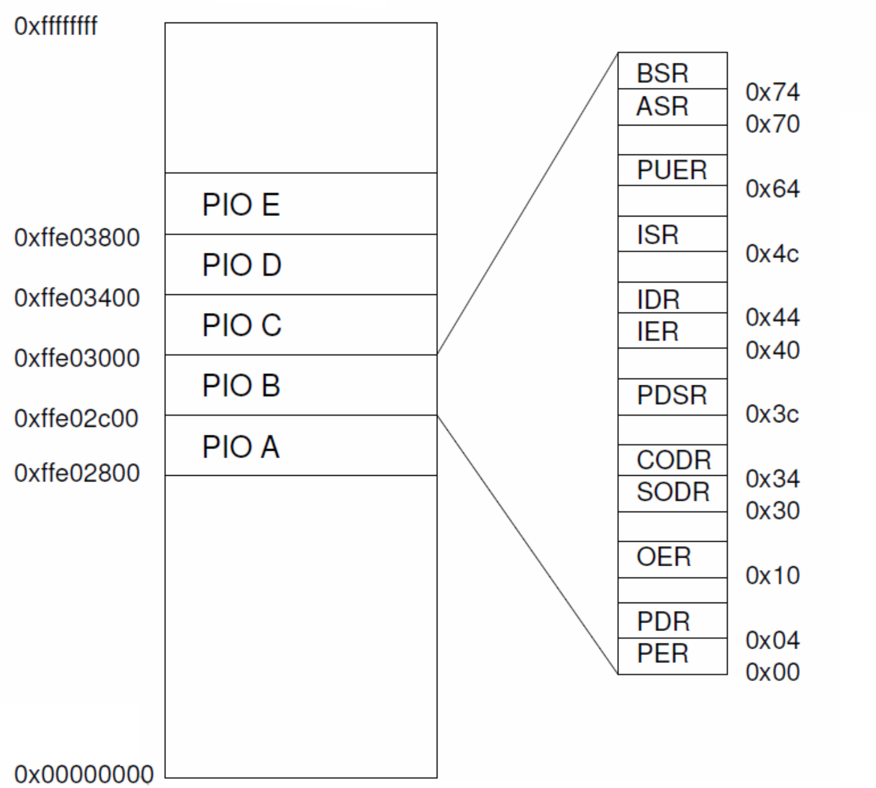
- I/O-controller: internally on the microcontroller
  - Controls the I/O pins of the microcontroller
  - General I/O pin:
    - Either input or output
    - Input: the program can read the value of I/O pin
    - Output: the program can set the value to I/O pin (low or high)

Configuration of PIO

- The microcontroller has memory mapped I/O:
  - Each I/O controller has a set of registers, each register is mapped on a specific address in the processor's address space
  - I/O controllers are controlled / programmed by writing to these registers
- There are 5 PIO ports, port A-E
  - Five sets of memory mapped registers
- Each PIO port has 32 bits
  - 32 I/O pins per PIO port
  - Each register has 32 bits, each bit corresponds to a given I/O pin on the microcontroller

PIO registers

Memory map

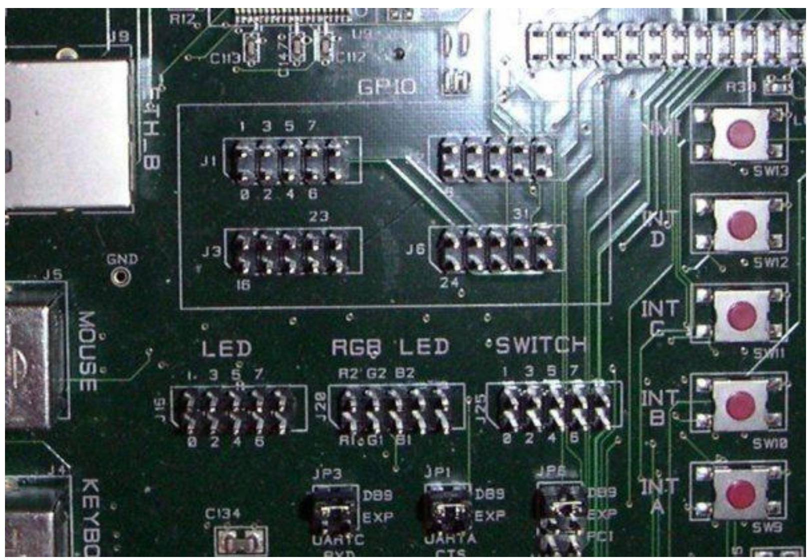


Register address:
base address + offset

PIO B,PUER:
0xffe02c00
+
0x64
=
0xffe02c64

GPIO on STK1000

- A selection of I/O pins goes to the GPIO connector
- Flat cables can connet GPIO to anything
- 1st Exercise: LEDs and buttons



PIO example: the use of buttons

- Connect buttons phisically to GPIO bus with flat cable
- Example: connect to GPIO 0-7 corresponding to PIO B pins 0-7
- In the program:
  - Enable I/O pins
    - Set bits 0-7 of register PIOB PER
  - Enable pull-up resistors
    - Set bits 0-7 of register PIOB PUER
  - To read the button status:
    - Read bits 0-7 of register PIOB PDSR

PIO example: the use of LEDs

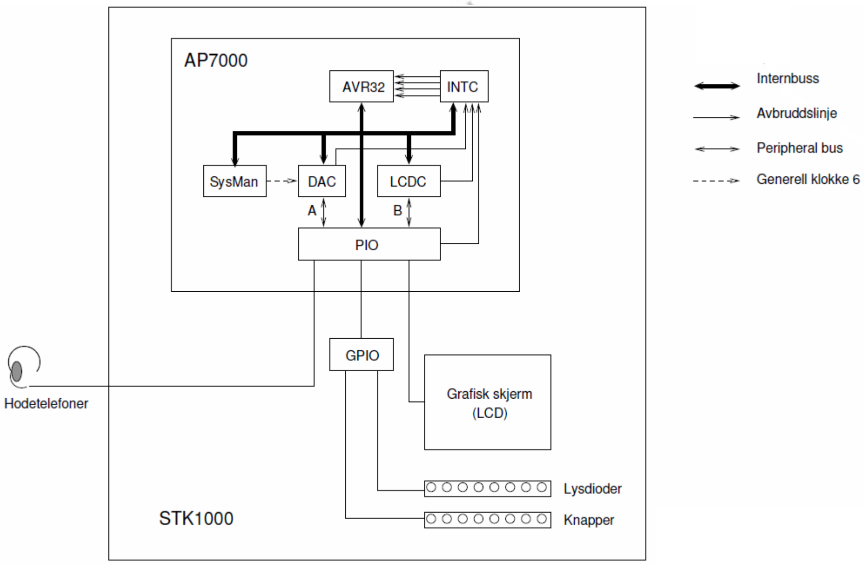
- Connect LEDs phisically to GPIO bus with flat cable
- Example: connect to GPIO 16-23 corresponding to PIO C pins 0-7
- In the program:
  - Enable I/O pins
    - Set bits 0-7 of register PIOC PER
  - Setting the I/O pins to be outputs
    - Set bits 0-7 of register PIOC OER
  - To turn off the LEDs
    - Set bits 0-7 of register PIOC CODR
  - To turn on the LEDs
    - Read bits 0-7 of register PIOC SODR

Interrupt

- Instead of polling I/O devices
- I/O units provide information when they want attention
- CPU saves the state of its parts and jumps to an interrupt routine
- Jumps back when the interrupt routine is completed

Interrupts on AVR32

- Four general interrupt lines
- Need many more
- Solved by having a separate interrupt controller INTC



Interrupt controller INTC

- Up to 64 groups of interrupts with up to 32 interrupt requests in each group
- Provides a maximum of 64*32 interrupts to the INTC
- It is hard connected
- Each group can be configured with
  - Autovector
  - Interrupt priority

Interrupt handling

- In case of interrupt: jump to interrupt routine
- The address of the interrupt routine is calculated as follows:
  – Interrupt routine address = EVBA | autovector
- EVBA: system register (Exception Vector Base Address, 32 bit)
- Autovector: offset from EVBA which AVR32 provides to INTC (14 bit)

Interrupt example

- Set up PIO B to provide interrupts
  - Program the interrupt routine
  - Set up PIO B to provide interrupts
    - Turn on the interrupts: register PIOB IER
    - Turn off the interrupts: register PIOB IDR
  - Determine and set EVBA
    - mtsr 4, r1
  - Calculate autovector and write to INTC the registry IPR14
  - Turn on the interrupts (delete the GM bit in the status register)
    - csrf 16

GNU tools

- From source code to executable programs
  (GCC) -> AS -> LD
- Automate with make
- Debugging: GDB
- Editor: Emacs (voluntary, well-integrated with GDB)

Assembling and linking

$ avr32-as -gstabs -o <objektfil> <assemblyfil>

$ avr32-ld -o <programfil> -l<bibliotek> <objektfiler>

Example:

$ avr32-as -gstabs -o foobar.o foobar.s

$ avr32-ld -o foobar.elf -lm foobar.o

make and Makefile

- Makefile contains commands to build the application
- Make reads the Makefile and performs the necessary commands

Example of Makefile

```
AS = avr32-as
ASFLAGS = -gstabs
LD = avr32-ld
# link: create ELF object files
eksempel.elf: eksempel.o
$(LD) eksempel.o -o eksempel.elf
# assembly: create object files from assembler files
eksempel.o: eksempel.s
$(AS) $(ASFLAGS) -o eksempel.o eksempel.s
# remove all auto generated files
.PHONY: clean
clean:
rm -rf *.o *.elf
```

NTNU – Trondheim
Norwegian University of
Science and Technology

GDB

- GDB: the GNU debugger
- Debug from PC via JTAGICE
- avr32gdbproxy -f 0,8Mb -a remote:1024
  - Start the proxy
- avr32-gdb <elf-programfile>
  - Start the GDB

GDB commands

target remote:1024	Connecting to the proxy
break <line number>	Set a break point
run	Run the program
bt	Trace back
si	Perform an instruction
c	Continue running
regs	Show registry content
help	Help

Emacs

- Key combinations: C=Ctrl, M=Alt
- Tutorial: C-h t (press Ctrl-h release, then press t)
- Some useful commands:
  - Open file: C-x C-f (find-file)
  - Save file: C-x C-s (save-buffer)
  - Exit: C-x C-c (save-buffers-kill-emacs)
  - Highlight text: C-<SPACE> (set-mark-command)
  - Cut selected test: C-w (kill-region)
  - Paste: C-y (yank)
  - Run an arbitrary command: M-x (execute-extended-command)

GDB in emacs

- Run the command gdb (M-x gdb RET)
- Enter a correct GDB command line
- GDB shows up as a separate buffer in Emacs
- GDB-one is connected to the source code buffer
  - Can set the break point directly in the source code: Cx <SPACE>
  - When GDB stops at a break point the line is highlighted in the source file

Help

- Where to find answers to all your wonder?
  - Excercise booklet
  - Documentation for the AVR32 and ATP32AP7000 (see course wiki or atmel.com)
  - The GNU tools: man-pages
  - google
  - Und.ass

Recommended actions

- Start ASAP
- Read the exercise booklet carefully
- Run the given code on the test card, make small changes and run the new one
- Try to control the LEDs
- Write the program without interrupts
- Add the interrupt handling

Lykke til

Where is the lab?

You can follow me now and I will show you!